Arm® Functional Fixed Hardware Specification

Platform Design Document

Non-Confidential

Version 1.2

Document number: DEN0048C

Contents

Release Information

The Change History table lists the changes made to this document.

Table 0-1 Change history

Date	Issue	Confidentiality	Change
17 April 2015	A	Non-Confidential	First release.
17 June 2020	B	Non-Confidential	Support for Processor Performance Monitoring through CPPC.
27 Sep 2022	C	Non-Confidential	▪ Support for Operation Regions. ▪ Fix typographical mistake in Table 8 header.

1 About this Document

This document provides the specification for Arm reserved uses of Functional Fixed Hardware for ACPI-based systems.

1.1 References

This document refers to the following documents.

Reference	Document Number	Title
[ACPI6.0]	ACPI 6.0	Advanced Configuration and Power Interface Specification version 6.0
[ACPI6.3]	ACPI 6.3	Advanced Configuration and Power Interface Specification version 6.3
[ACPI6.4]	ACPI 6.4	Advanced Configuration and Power Interface Specification version 6.4.
[ACPI6.5]	ACPI 6.5	Advanced Configuration and Power Interface Specification version 6.5.
[Arm ARM]	DDI0487	Arm Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. See https://developer.arm.com/documentation/ddi0487/latest
[FF-A]	DEN0077	Arm Firmware Framework for Arm A-profile. See https://developer.arm.com/documentation/den0077/latest/
[PSCI]	DEN0028	Power State Coordination Interface. See https://developer.arm.com/documentation/den0022/latest/
[SMCCC]	DEN0028	Arm SMC Calling Convention. See https://developer.arm.com/documentation/den0028/latest/

1.2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AMU	Activity Monitor Unit
CPC	Continuous Performance Control
CPPC	Collaborative Processor Performance Control
FFH	Functional Fixed Hardware. This refers to *software* (SW) operations that replace a *hardware* (HW) function.
LPI	Low Power Idle
MRS	Move to Arm register from System Register Instruction
OSPM	Operating System-directed configuration and Power Management

1.3 Feedback

Arm welcomes feedback on its documentation.

1.3.1 Feedback on this manual

If you have comments on the content of this manual, send an e-mail to errata@arm.com. Give:

- The title.
- The document and version number, DEN0048C.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1.4 Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

2 Introduction

This document provides a specification for *Functional Fixed Hardware* (FFH) in Arm-based systems that use the *Advanced Configuration and Power Interface* (ACPI) specification.
At the time of writing, the three use cases are:

- Idle Management and *Low Power Idle* (LPI) states. See [ACPI6.0].
- Performance Management and Collaborative Processor Performance Control (CPPC). See Section 8.4.7 of [ACPI 6.3]
- Operation Regions. See [ACPI6.5].

3 Use Cases

3.1 Idle management and Low Power Idle states

ACPI 6.0 [ACPI6.0] introduces *Low Power Idle* (LPI) states, which allow an operating system to manage the power states of the processor power domain hierarchy. This section describes how FFH is used in Arm-based systems to allow the operating system to discover:

- The entry method into a low power state.
- How to collect power state residency, and usage count statistics.

This section also defines the flags used in an LPI state object to describe the architectural context that is lost when the LPI state is entered.

3.1.1 FFH Usage in LPI state entry methods

ACPI ASL uses the Register keyword to define HW register addresses, or SW functions, when using FFH. The Register keyword has the following format:

Register (*AddressSpaceKeyword, RegisterBitWidth, RegisterBitOffset, RegisterAddress, AccessSize, DescriptorName*)

For further information, see section 19.6.108 of [ACPI6.0]. Registers are used to specify one of the following two entry methods into an LPI state:

- A *Wait For Interrupt* (WFI) instruction.
- A PSCI `CPU_SUSPEND` call. In this case, the entry method provides a way of describing the `power_state` parameter of the `CPU_SUSPEND` call as specified by [PSCI].

When using FFH to describe LPI entry methods, the register field entries must be set as follows:

- *AddressSpaceKeyword* must be set to 0x7f. This denotes usage of the FFH address space.
- *RegisterBitWidth* must be set to 32.
- *RegisterBitOffset* must be set to 0.
- *AccessSize* must be set to 3 (Dword).
- WFIs states must be represented in the _LPI objects of processors. In the WFI case, the *RegisterAddress* in the entry method of the LPI state has the following format:

Bits[63:32]	Bits[31:0]
0x00000000	0xFFFFFFFF

- A PSCI `power_state` parameter is represented in the *RegisterAddress* field as follows:

Bits[63:32]	Bits[31:0]
0x00000000	PSCI `power_state` parameter for `CPU_SUSPEND` call. See section 5.4 of [PSCI] for more details.

- For LPI entry methods, all other possible encodings of *RegisterAddress*, *RegisterBitWidth, RegisterBitOffset,* and *AccessSize* where 0x7f is used for the *AddressSpaceKeyword* are reserved for future use.
- *DescriptorName* is optional. See section 19.6.108 of [ACPI6.0] for further details.

When the OS is working in OS Initiated mode, as defined by both [PSCI] and [ACPI6.0], the OS must regard cores using the WFI state as being in a running state, for the purposes of last man tracking. To enter OS Initiated mode, the OS must use the `PSCI_SET_SUSPEND_MODE` call described in sections 5.1.19 and 5.20 of [PSCI].

Appendix A provides a description and examples of how PSCI power states are composed from LPI entry methods and _LPI LevelID. The examples cover platform coordinated and OS Initiated systems.

3.1.2 FFH Usage in LPI residency and usage counter registers

PSCI1.0 introduces the `PSCI_STAT_RESIDENCY` and `PSCI_STAT_COUNT` functions. For systems that implement these functions, ASL Registers in the FFH space can be used for the residency and usage counter register fields of LPI states. This allows matching those registers to a PSCI call. In this case, the format of the Register provided is as follows:

- *AddressSpaceKeyword* must be 0x7f. This denotes usage of the FFH address space.
- *RegisterBitWidth* must be 32.
- *RegisterBitOffset* must be 0.
- *AccessSize* must be 3 (Dword).
- For both the residency and usage counter registers, the *RegisterAddress* field must have the following encoding:

Bits[63:32]	Bits[31:0]
0x00000000	PSCI `power_state` parameter for `PSCI_STAT_*` functions. See section 5.21.1 of [PSCI].

- For residency and usage counter registers of LPI entries, all other possible encodings of *RegisterAddress*, *RegisterBitWidth, RegisterBitOffset,* and *AccessSize* where 0x7f is used for the *AddressSpaceKeyword* are reserved for future use.
- *DescriptorName* is optional and can be omitted. See section 19.6.108 of [ACPI6.0] for further details.

In the case of the residency counter register, this encoding instructs the processor driver of the OS to issue a `PSCI_STAT_RESIDENCY` call. The residency counter frequency of the LPI must be set to 1000000, to indicate that the count is in microseconds.

In the case of the usage counter register, this encoding instructs the processor driver of the OS to issue a `PSCI_STAT_COUNT` call.

Note: The OSPM must use the `PSCI_FEATURES` API to ensure that `PSCI_STAT_RESIDENCY` and `PSCI_STAT_COUNT` are provided by the PSCI implementation. See [PSCI] for more details.

3.1.3 Save and restore flags

LPI states provide an architectural context loss flags field that can be used to describe the context that might be lost when an LPI state is entered. For Arm-based systems, the flags have the following format:

Table 2 Arm Architecture context loss flags

Flag	Bit offset	Bit length	Description
Core context Lost	0x0	0x1	All core context is lost. This includes: • General purpose registers. • Floating point and SIMD registers. • System registers, include the System register based

			generic timer for the core. • Debug register in the core power domain. • PMU registers in the core power domain. • Trace register in the core power domain.
Trace context loss	0x1	0x1	Trace registers outside of the core power domain are lost.
GICR	0x2	0x1	GIC Redistributor logic.
GICD	0x3	0x1	GIC Distributor logic.
Reserved	0x4	0x1c	Reserved must be zero.

3.2 Performance management and Collaborative Processor Performance Control

ACPI uses collaborative processor performance control (CPPC) to define an abstracted and flexible mechanism for Operating System-directed configuration and Power Management (OSPM) to collaborate with an entity in the platform to manage the performance of a logical processor. In this scheme, the platform entity is responsible for creating and maintaining a performance definition that backs a continuous, abstract, unit-less performance scale. During runtime, OSPM requests desired performance on this abstract scale and the platform entity is responsible for translating the OSPM performance requests into actual hardware performance states. The control mechanisms are abstracted by the Continuous Performance Control (_CPC) object method, which describes how to control and monitor processor performance in a generic manner.

This section describes how FFH is used in Arm-based systems to monitor the performance of a logical processor using the Activity Monitor Unit (AMU).

3.2.1 FFH Usage in _CPC object to monitor processor performance

ACPI ASL uses the Register keyword to define HW register addresses, or SW functions, when using FFH. The Register keyword has the following format:

Register (*AddressSpaceKeyword, RegisterBitWidth, RegisterBitOffset, RegisterAddress, AccessSize, DescriptorName*)

For further information, see section 19.6.114 of [ACPI6.3].

Arm cores which implement the Activity Monitor Unit (AMU), can use ASL Registers in the FFH space to specify the following two fields of the _CPC package to monitor the performance of a logical processor:

- *DeliveredPerformanceCounterRegister*
- *ReferencePerformanceCounterRegister*

ASL registers in the FFH space allow to map:

- a read from the *DeliveredPerformanceCounterRegister* to a corresponding read from the AMU event counter that counts cycles at core frequency (AMEVCNTR0_EL0[0]).
- a read from the *ReferencePerformanceCounterRegister* to a corresponding read from the AMU event counter that counts cycles at constant frequency (AMEVCNTR0_EL0[1]).

The AMU Event Counters can be read through a system register read using the MRS instruction. Details on AMU event counters can be found in the [Arm ARM].

When FFH is used to describe the *DeliveredPerformanceCounterRegister* and the *ReferencePerformanceCounterRegister* fields of the _CPC package, the register field entries must be set as follows:

- *AddressSpaceKeyword* must be set to 0x7f. This denotes usage of the FFH address space.
- *RegisterBitWidth* must be set to 64.
- *RegisterBitOffset* must be set to 0.
- *AccessSize* must be set to 4 (Qword).
- *RegisterAddress* must be set to
    - 0x0 for *DeliveredPerformanceCounterRegister*.
    - 0x1 for *ReferencePerformanceCounterRegister*.
- *DescriptorName* is optional. See section 19.6.114 of [ACPI6.3] for further details.

All other possible encodings of *RegisterAddress*, *RegisterBitWidth*, *RegisterBitOffset*, and *AccessSize* where 0x7f is used for the *AddressSpaceKeyword* are reserved for future use.

Appendix B describes the fields of the _CPC package that should be provided to monitor and control processor performance through CPPC for Arm-based systems.

3.3 Operation Regions

ACPI Control Methods can read and write data to locations in address spaces by using the *Field* operator to declare a data element within an entity known as an “Operation Region”. Control methods must have exclusive access to any address accessed via fields declared in Operation Regions. FFH can be used to describe an Operation Region space. For more details on Operation Regions, see [ACPI6.5].

This section describes how FFH can be used in Arm-based systems to describe Operation Region spaces and the expected behavior when reading or writing FFH Operation Region fields.

3.3.1 FFH usage in Operation Region Space for Secure Monitor Calls (SVC) and Hypervisor Calls (HVC)

FFH Operation Region space can be used to trigger SMC or HVC calls, using the Arm SMC Calling Convention (SMCCC). For more details see [SMCCC]. The choice of conduit (SMC or HVC) is implementation defined and outside the scope of the FFH Operation Region definition.

The syntax of the *OperationRegion* term is described below:

```
OperationRegion (
        RegionName,   //NameString
        RegionSpace,  //RegionSpacekeyword
        Offset,       //Integer
        Length        //Integer
)
```

Where:

- *RegionName* specifies a name for this Operation Region (for example, “AFFH”).
- *RegionSpace* must be set to 0x7F (FFixedHW). This denotes usage of the FFH address space.
- *Offset* is used to identify the functionality offered by this FFH address space. It must be set to one of the following values:
    - *0x0* to indicate usage of 32-bit calling convention.
    - *0x1* to indicate usage of 64-bit calling convention.
    - All other values are reserved.
- *Length* specifies the higher of either the number of argument bytes, or the number of result bytes, exchanged over the SMC/HVC call. It is used to interpret the number of registers to use for the call. The registers are listed sequentially, beginning from, and including, the Function Identifier (W0/R0). For example, when Length is 12, registers W0, W1, W2 are used in a SMC32 call.

  Length must be set to one of the following values, where ‘*N*’ denotes the higher of either the number of argument registers or the number of result registers used for the call:
    - If *Offset* is 0 (32 bit calling convention):
        - *Length* must be 4**N*, where $1 \le N \le 8$.
    - If *Offset* is 1 (64 bit calling convention):
        - *Length* must be 8**N*, where $1 \le N \le 18$.
    - All other values of Length are reserved.

FFH operation regions are only accessible via the *Field* term. The syntax of the Field term is described below:

```
Field (
        RegionName,   //NameString
        AccessType,   //AccessTypeKeyword
        LockRule,     //LockRuleKeyword
        UpdateRule    //UpdateRuleKeyword
) {FieldUnitList}
```

Where:

- *RegionName* specifies the operation region name previously defined for this FFH Operation Region.
- *Accesstype* must be set to *BufferAcc*. This denotes usage of the FFH address space.
- *LockRule* indicates if access to this Operation Region requires acquisition of the Global Lock for synchronization. This field must be set to NoLock, as required for HW-reduced ACPI platforms.
- *UpdateRule* is not applicable for FFH Operation Regions since each virtual register is accessed in its entirety.
- *FieldUnitList* comprises of a single element representing the entire Operation Region and the total data exchanged over the SMC or HVC call. Writing to this field element triggers the call.
  This means that each argument register cannot be broken down within the field definition. Access to argument registers can be done only outside of the field definition. This limitation is imposed both to simplify the interface and to maintain consistency with the model defined by the Arm SMC Calling Convention [SMCCC].

An example Operation Region and Field definition is shown below. Here we declare 5 argument registers used for a SMC32 call.

```
OperationRegion(AFFH, FFixedHW, 0, 20)

Field (AFFH, BufferAcc, NoLock, Preserve) { SMCC, 160 }
```

Appendix C describes detailed examples of FFH Operation Region Usage in various calling modes.

3.3.1.1 Restrictions

The usage of FFH Operation Region space for SMC or HVC calls is subject to the following restrictions:

- All calls must be compliant with the Arm SMC Calling Convention.
- Function Identifiers from only the following ranges are allowed:
  - SMCCC SiP Service call range.
  - SMCCC OEM Service call range.
  - FF-A specific Function Identifiers. For more details see [FF-A].
- It is recommended to use Fast Service Calls to simplify the usage model. The call appears to be atomic from the perspective of the calling PE and returns when the requested operation has completed. If Yielding Service Calls are used, then it is the responsibility of the caller to safely resume the operation post pre-emption.

3.3.1.2 Error handling

There might be circumstances where the SMC or HVC call cannot be triggered when the Operation Region field element is written to. Examples could be when the Function Identifier (R0/W0) of the SMC call is malformed, or when the restrictions specified in Section 3.3.1.1 are violated. In such cases, an appropriate negative integer return code from the Arm SMC Calling Convention [SMCCC] can be used to specify the error condition in:

- The first 32 bits (W0) of the Operation Region field element when 32-bit calling convention is used.
- The first 64 bits (X0) of the Operation Region field element when 64-bit calling convention is used.

Appendix A PSCI state composition from LPI states

Section 8.4.4.3.4 of [ACPI6.0] describes how entry methods of local LPI states are composed to produce the final command that is issued to platform firmware to enter a composite power state. This appendix describes how this composition takes place in Arm systems that provide a PSCI implementation. In such systems, the composition results in the value of the `power_state` parameter that is passed to a `CPU_SUSPEND` call to enter the composite state.

Each LPI state provides an entry method field that is used to determine the PSCI `power_state` parameter to be used with a `CPU_SUSPEND` call. The parameter is composed through the following steps:

1. The OS extracts an initial base `power_state` parameter from the lower 32 bits of the *RegisterAddress* field of the entry method of a processor LPI. For processors, entry methods must be a register that adheres to the definition provided in section see 3.1.1.
2. If the composite power state selected by an OS affects power levels above the processor, the OS must walk the LPI states defined in processor containers above the processor. For the LPI states in those containers, the entry method can be an integer or a register:
    a. If the entry method is an integer value, then the base `power_state` parameter obtained in step 1 must have this integer value added to it.
    b. If the entry method defined is a register, then the lower 32 bits of the *RegisterAddress* field becomes the new base `power_state` parameter

   This process is repeated across the LPIs that form the target power state. For OS and PSCI firmware working in platform coordinated mode, the base `power_state` parameter obtained in steps 1 and 2 forms the final `power_state` parameter that is passed to a `CPU_SUSPEND` call.
3. For OS and PSCI firmware working in OS Initiated mode, the OS must indicate the power level in which it observes that the calling processor is the last to go idle. To do so, the OS adds the value of the LevelID field of the _LPI object [ACPI6.0], defined at the appropriate power level, to the base `power_state` parameter, to form the final `power_state` argument for the `CPU_SUSPEND` call.

The steps are described in the following pseudocode:

```
// Initially LPIx points to a processor-level LPI state
LPIx = ChooseLPIStateForLevel(LevelOf(CurrentProcessor),NULL)

power_state = LPIx.EntryMethod.Address

if power_state == uint_64(-1) // WFI case
        doWFI()
        return

for level = Parent(CurrentProcessor) to system
        LPIx = ChooseLPIStateForLevel(level,LPIx)

        If LocalState == Run
                break

        EM = LPIx.EntryMethod

        if IsInteger(EM)
                power_state = power_state+ZeroExtend(IntegerValue(EM))
        else
                power_state = EM.Address
```

```
If IdleMode == OS_Initiated
      LastProc = ProcessorContainerWhereCallingIsLastToIdle()
      LastProcLPI = LastProc.LPI
      power_state = power_state+LastProcLPI.LevelID

doCPU_SUSPEND(power_state)
return
```

The following sections provide examples of how power states can be represented in systems using the original StateID `power_state` parameter format, and the extended StateID `power_state` parameter format. The first example applies to PSCI 0.2 or PSCI 1.0, while the second applies only to PSCI 1.0.

A.1 Original StateID `power_state` parameter format: PSCI0.2 or above

PSCI 0.2 supports only the original `power_state` parameter format with a 16-bit StateID field. See [PSCI] for further details. Figure 1 shows an example system.

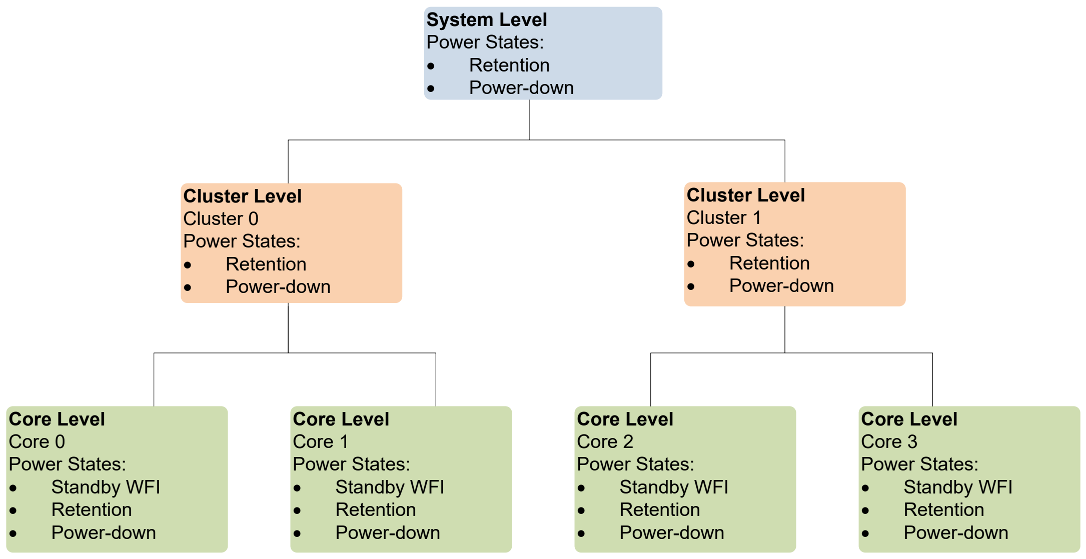

Figure 1 Example system

Figure 1 shows an example system composed of three power levels, core, cluster, and system. For each power level, the local power states are shown. Our example system PSCI supports the composite power states shown in Table 3.

Table 3 Supported power states in the example system and PSCI `power_state` parameter encoding in original StateID format

Composite power state			
Core local state	Cluster local state	System local state	PCI `power_state` parameter
Retention	Run	Run	0x00000001
Power-down	Run	Run	0x00010002
Retention	Retention	Run	0x01000011

Power-down	Retention	Run	0x01010012
Power-down	Power-down	Run	0x01010022
Retention	Retention	Retention	0x02000111
Power-down	Retention	Retention	0x02010112
Power-down	Power-down	Retention	0x02010122
Power-down	Power-down	Power-down	0x02010222

Note: Standby WFI is not listed as this state is entered via a WFI instruction and not via PSCI.

Table 3 shows the supported power states and their PSCI encoding. The example uses the following encoding:

- The power state StateID field, bits[15:0], is broken into sections that indicate the local power state of the core, cluster, and system. The possible values for each level are:

  0: The entity is in the Run state.

  1: The entity is in the Retention state.

  2: The entity is in is the Power-down state.

  The bit fields of the StateID field are as follows:

  - Bits[1:0] indicate the local power state at the core level.
  - Bits[5:4] indicate the local power state at the cluster level.
  - Bits[9:8] indicate the local power state at the system level.
  - Bits[13:12] are used by an OS working in OS Initiated mode to indicate the level in which the calling core is the last man. The OS might write:
    - 0x0 if the core is not the last man at cluster or system level.
    - 0x1 if the core is the last man at cluster level.
    - 0x2 if the core is the last man in the system.
  - All other bits are set to zero.
- As indicated in [PSCI], the PowerLevel field, bits[25:24], of the `power_state` parameter reflects the highest power level affected by the state.
- As indicated in [PSCI], the StateType bit, bit 16, of the `power_state` parameter is set whenever the local state of the core is Power-down.

Table 4 shows how this PSCI encoding for power states can be represented in LPI entry methods.

Table 4 LPI States

Core Level – LevelID 0				
State	Description	Entry Method	Entry method value	Enabled Parent State
LPI1	Standby WFI	Register	*RegisterAddress* `0xffffffff`	0 // Does not enable states in cluster or system levels.
LPI2	Retention	Register	*RegisterAddress* `0x00000001`	1 // Enables cluster retention.
LPI3	Power-down	Register	*RegisterAddress* `0x00010002`	2 // Enables cluster retention and power-down.

Cluster Level – LevelID 0x1000				
LPI1	Retention	Integer	`0x01000010`	1 // Enables system retention.
LPI2	Power-down	Integer	`0x01000020`	2 // Enables system retention and power-down.
System Level – LevelID 0x2000				
LPI1	Retention	Integer	`0x01000100`	0 // N/A.
LPI2	Power-down	Integer	`0x01000200`	0 // N/A.

Table 5 demonstrates how the entry methods for the LPI states are combined to produce the correct PSCI power-state parameters.

Table 5 PSCI `power_state` composition from LPI states

Composite power state			
Core local state	Cluster local state	System local state	PCI `power_state` parameter
LPI2: Retention 0x00000001	+ LPI0: Run + 0	+ LPI0: Run + 0	= 0x00000001
LPI3: Power-down 0x00010002	+ LPI0: Run +0	+ LPI0: Run +0	= 0x00010002
LPI2: Retention 0x00000001	+ LPI1: Retention + 0x01000010	+ LPI0: Run + 0	= 0x01000011
LPI3: Power-down 0x00010002	+ LPI1: Retention + 0x01000010	+ LPI0: Run + 0	= 0x01010012
LPI3: Power-down 0x00010002	+LPI2: Power-down +0x01000020	+ LPI0: Run + 0	= 0x01010022
LPI2: Retention 0x00000001	+ LPI1: Retention + 0x01000010	+ LPI1: Retention +0x01000100	= 0x02000111
LPI3: Power-down 0x00010002	+ LPI1: Retention + 0x01000010	+ LPI1: Retention +0x01000100	= 0x02010112
LPI3: Power-down 0x00010002	+LPI2: Power-down + 0x01000020	+ LPI1: Retention +0x01000100	= 0x02010122
LPI3: Power-down 0x00010002	+LPI2: Power-down +0x01000020	+LPI2: Power-down +0x01000200	= 0x02010222

A system using OS Initiated mode adds the LevelID field value to the `power_state` parameter obtained by combining the LPI state entry methods:

- If a core is not the last man in the cluster or system, no addition is required.
- If a core is the last man in the cluster, the LevelID value of 0x1000 must be added to the `power_state` parameter.
- If a core is the last man in the system, the LevelID value of 0x2000 must be added to the `power_state` parameter.

A.2 Extended StateID `power_state` parameter format: PSCI 1.0 or above

PSCI 1.0 introduces an optional new format for the `power_state` parameter with an extended StateID field. This provides more flexibility to PSCI implementers on how to encode power states. Referring to the example system from Figure 1, Table 6 shows a possible PSCI 1.0 encoding using the extended ID format.

Table 6 Supported power states in the example system and PSCI `power_state` parameter encoding in the extended StateID format

Composite power state			
Core local state	Cluster local state	System local state	PCI `power_state` parameter
Retention	Run	Run	0x00000001
Power-down	Run	Run	0x40000002
Retention	Retention	Run	0x00000011
Power-down	Retention	Run	0x40000012
Power-down	Power-down	Run	0x40000022
Retention	Retention	Retention	0x00000111
Power-down	Retention	Retention	0x40000112
Power-down	Power-down	Retention	0x40000122
Power-down	Power-down	Power-down	0x40000222

Note: Standby WFI is not listed as this state is entered via a WFI instruction and not via PSCI.

Table 6 shows the supported power states and their PSCI encoding. The example uses the following encoding:

- StateID:
  - Bits[1:0] represent the power state of the core power level, where 0 represents the Run state, 1 the Retention state, and 2 the Power-down state.
  - Bits[5:4] represent the power state of the cluster power level. 0 represents the Run state, 1 the Retention state, and 2 the Power-down state.
  - Bits[9:8] represent the power state of the system power level. 0 represents the Run state, 1 the Retention state, and 2 the Power-down state.
  - Bits[25:24] are used by an OS working in OS Initiated mode to indicate the level in which the calling core is the last man. The OS might write:
    - 0x0 if the core is not the last man at cluster or system level.
    - 0x1 if the core is the last man at cluster level.
    - 0x2 if the core is the last man in the system.
  - All other bits are set to zero

- As indicated in [PSCI] the StateType bit, bit 30, of the `power_state` parameter is set whenever the local state of the core is Power-down.

Table 7 shows how this PSCI encoding for power states can be represented in LPI entry methods.

Table 7 LPI States

Core Level – LevelID 0				
State	Description	Entry Method	Entry method value	Enables Parent State
LPI1	Standby WFI	Register	*RegisterAddress* `0xffffffff`	0 // Does not enable states in cluster or system levels.
LPI2	Retention	Register	*RegisterAddress* `0x00000001`	1 // Enables cluster retention.
LPI3	Power-down	Register	*RegisterAddress* `0x40000002`	2 // Enables cluster retention and power-down.
Cluster Level – LevelID 0x01000000				
LPI1	Retention	Integer	`0x00000010`	1 // Enables system retention.
LPI2	Power-down	Integer	`0x00000020`	2 // Enables system retention and power-down.
System Level – LevelID 0x02000000				
LPI1	Retention	Integer	`0x00000100`	0 // N/A.
LPI2	Power-down	Integer	`0x00000200`	0 // N/A.

Table 8 demonstrates how the entry methods for the LPI states are combined to produce the correct PSCI power-state parameters:

Table 8 PSCI `power_state` composition from LPI states

Composite power state			
Core local state	Cluster local state	System local state	PSCI `power_state` parameter
LPI2: Retention 0x00000001	+ LPI0: Run + 0	+ LPI0: Run + 0	= 0x00000001
LPI3: Power-down 0x40000002	+ LPI0: Run +0	+ LPI0: Run +0	= 0x40000002
LPI2: Retention 0x00000001	+ LPI1: Retention + 0x00000010	+ LPI0: Run + 0	= 0x00000011
LPI3: Power-down	+ LPI1: Retention	+ LPI0: Run	

0x40000002	+ 0x00000010	+ 0	= 0x40000012
LPI3: Power-down 0x40000002	+LPI2: Power-down + 0x00000020	+ LPI0: Run + 0	 = 0x40000022
LPI2: Retention 0x00000001	+ LPI1: Retention + 0x00000010	+ LPI1: Retention + 0x00000100	 = 0x00000111
LPI3: Power-down 0x40000002	+ LPI1: Retention + 0x00000010	+ LPI1: Retention + 0x00000100	 = 0x40000112
LPI3: Power-down 0x40000002	+LPI2: Power-down + 0x00000020	+ LPI1: Retention + 0x00000100	 = 0x40000122
LPI3: Power-down 0x40000002	+LPI2: Power-down + 0x00000020	+LPI2: Power-down + 0x00000200	 = 0x40000222

A system using OS Initiated mode also adds the value of the LevelID field to the `power_state` parameter that is obtained by combining the LPI state entry methods:

- If a core is not the last man in the cluster or system, no addition is required.
- If a core is the last man in the cluster, the LevelID value of 0x01000000 must be added to the `power_state` parameter.
- If a core is the last man in the system, the LevelID value of 0x02000000 must be added to the `power_state` parameter.

Appendix B Processor performance monitoring and control in heterogenous systems

To implement processor performance management through CPPC, the platform needs to specify the fields of the _CPC package as outlined in Section B.1 and B.2.

B.1 Monitoring Processor Performance through CPPC

To monitor processor performance through CPPC, the platform should specify the following fields of the _CPC package:

- *DeliveredPerformanceCounterRegister*
- *ReferencePerformanceCounterRegister*
- *ReferencePerformance*
- *PerformanceLimitedRegister* - This *Register* returns zero when read if the hardware does not support signaling performance limited events.

The platform must use the same performance scale for all processors in the system such that any two processors running the same workload at the same performance level will complete in approximately the same time. OSPM reads the set of performance counters from the Reference Performance Counter Register and the Delivered Performance Counter Register to determine the actual performance level delivered over time (refer Section 8.4.7.1.3.1 of [ACPIv6.3]). OSPM calculates the delivered performance over a given time period by taking a beginning and ending snapshot of both the reference and delivered performance counters, and calculating:

$$\textit{delivered performance} = \textit{reference performance} \quad \text{X} \quad \frac{\Delta \textit{delivered performance counter}}{\Delta \textit{reference performance counter}}$$

ReferencePerformance indicates the performance level at which the Reference Performance Counter accumulates. When the reference performance counter and the delivered performance counter are incrementing at the same rate, the processor is running at *ReferencePerformance.*

When *DeliveredPerformanceCounterRegister* and *ReferencePerformanceCounterRegister* are mapped to AMU event counters (refer Section 3.2.1), *ReferencePerformance* can be mathematically derived as follows:

$$\textit{ReferencePerformance} = R_{const} \times R_{uarch} \times k, \text{ where}$$

- *Rconst* is the fixed rate at which the AMU event counter that counts cycles at constant frequency (AMEVCNTR0_EL0[1]) accumulates. This is equal to the rate at which the system counter accumulates.
- *Ruarch* is a micro-architectural multiplier. It is a first-order approximation of the relative performance of the logical processor.
  - On platforms where performance characteristics of all the logical processors are identical at the same processor frequency, *Ruarch* is set to 1.
  - On platforms with heterogeneous processors, the performance characteristics of all logical processors are not always identical at the same processor frequency. Such heterogeneity can be the result of micro-architectural differences, like in Arm big.LITTLE systems. In this case, as mandated by Section 8.4.7 of [ACPIv6.3], the platform must synthesize a performance scale that adjusts for differences in processors, such that any two processors running the same workload at the same performance level will complete in approximately the same time. *Ruarch* is the multiplier that normalizes for differences in performance between logical processors running at identical processor frequency. It is a first order approximation. *Ruarch*

can be measured by running a representative benchmark on each logical processor. An example of *Ruarch* can be the DMIPS/MHz of the logical processor.

- *k* is the multiplier that normalizes *Rconst* x *Ruarch* to the performance scale synthesized by the platform. *k* should be the same across all logical processors in the system.

An example of a performance scale in a simplified big.LITTLE system is shown in Figure 2. In this example, the platform synthesizes a performance scale between 0 and 100. For simplicity we assume that the performance of the cores scale linearly with core frequency. The AMU event counter which counts cycles at constant frequency is running at 1000MHz (=*Rconst*). The LITTLE cores can run at frequencies ranging between 250MHz to 1250MHz with *Ruarch* = 1. The big cores can run at frequencies ranging between 500MHz to 2000MHz with *Ruarch* = 2. *Ruarch* of 2 signifies that the performance of the big cores is 2x the performance of the LITTLE cores at the same core frequency. *k* is chosen to be 0.025 to synthesize a performance scale between 0 and 100. When *k* is 0.025, *ReferencePerformance* is 25 for the LITTLE core and 50 for the big core.

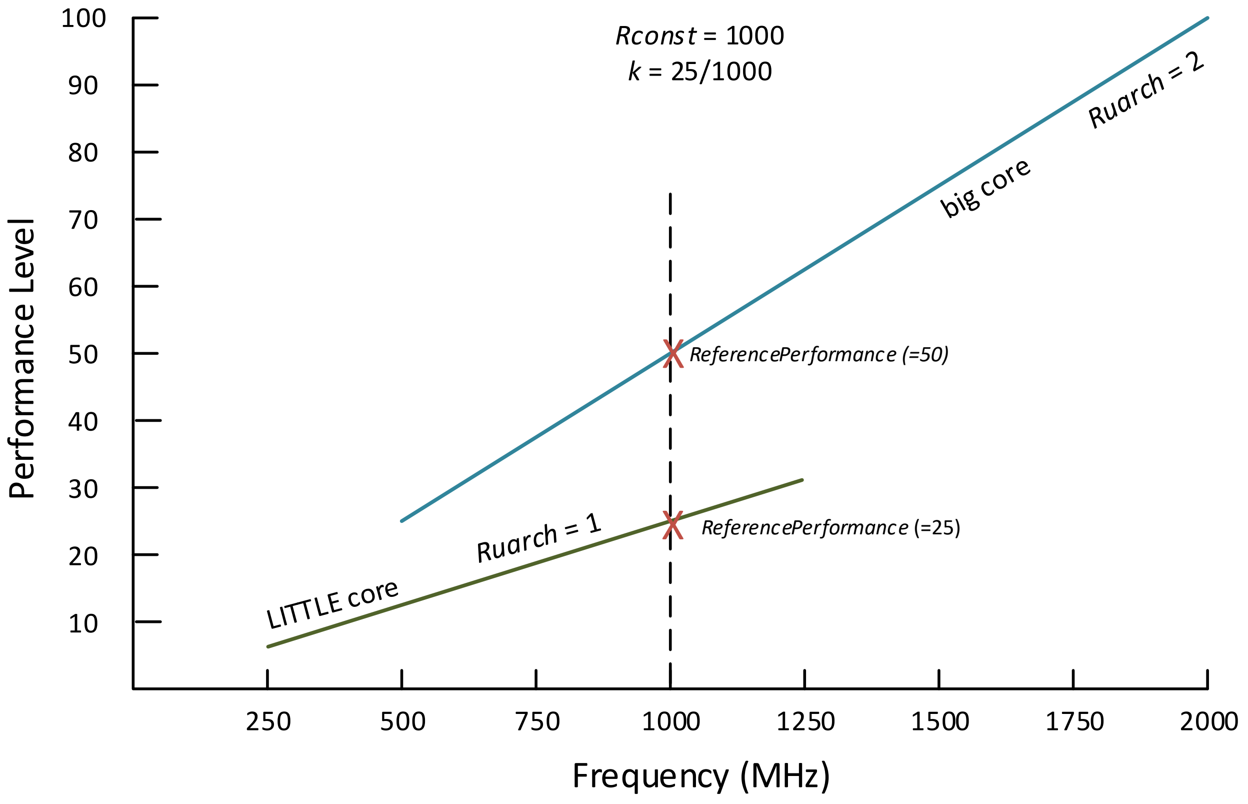

Figure 2: Example performance scale setup in a big.LITTLE system.

B.2 Controlling Processor Performance through CPPC

To control processor performance through CPPC, the platform should report the performance capabilities/thresholds and performance controls as specified in the sections below.

B.2.1 Performance Capabilities/Thresholds

Platforms should specify the following performance capabilities/threshold fields of the _CPC package as a *Register* (refer Section 8.4.7.1.1 of [ACPIv6.3]):

- *HighestPerformance*
- *NominalPerformance*
- *LowestNonlinearPerformance*
- *LowestPerformance*

Platforms should also provide the following optional fields of the _CPC package as a *Register* or *DWORD*, when they must work with Operating Systems which need to report CPU frequency, and there is no alternate mechanism to discover this information. For more details, refer Section 8.4.7.1.1.7 of [ACPIv6.3].

- *LowestFrequency*
- *NominalFrequency*

Figure 3 shows the performance thresholds of the same big.LITTLE system as referred in Figure 2.

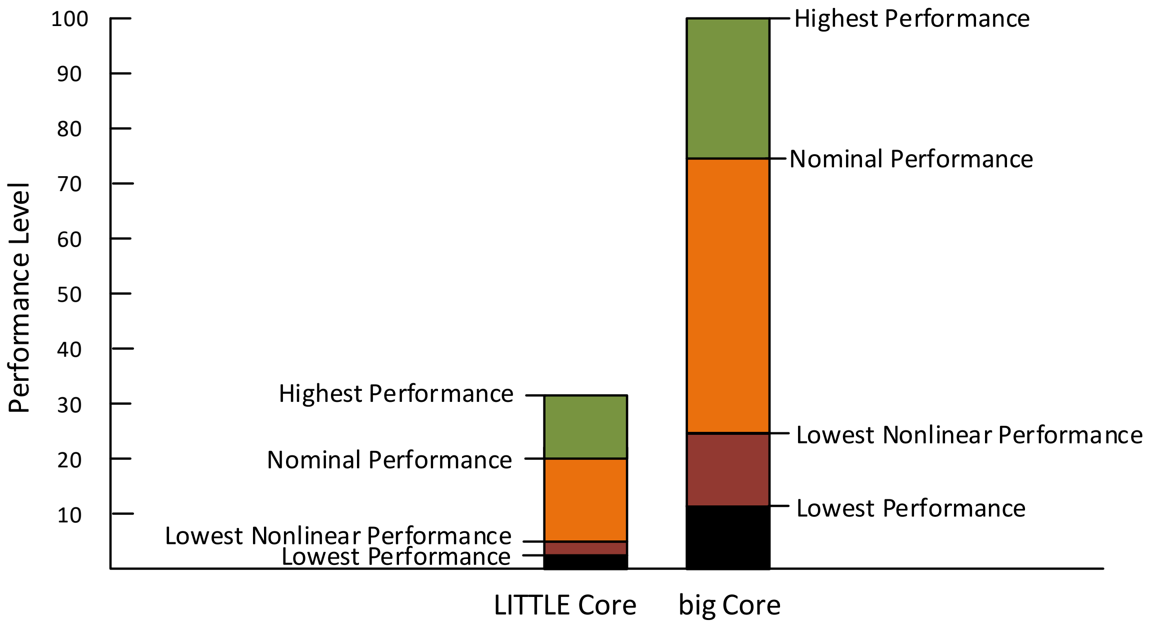

Figure 3: Example performance thresholds in a big.LITTLE system with same characteristics as Figure 2.

B.2.2 Performance Controls

Platforms should specify the following performance control field of the _CPC package as a *Register* (refer Section 8.4.7.1.2 of [ACPIv6.3]):

- *DesiredPerformanceRegister*

All other fields of the _CPC package can be optionally implemented by the platform depending on its capabilities.

An example implementation of _CPC for the system corresponding to Figure 3 is shown below.

```
DefinitionBlock("DsdtTable.aml", "DSDT", 1, "ARMLTD", "ARM-Example", 1) {
 Scope(_SB) {
  Device(CPU1) {
    Name(_HID, "ACPI0007")
    Name(_UID, 1)
    Name(_CPC, Package()
    {
       23,     // NumEntries
       3,      // Revision
       32,     // Highest Performance
       20,     // Nominal Performance
       6,      // Lowest Nonlinear Performance
       3,      // Lowest Performance
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Guaranteed Performance Register
       ResourceTemplate(){Register(SystemMemory, 0x20, 0, 0xA0051000, 0x3)}, // Desired Perf Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Minimum Performance Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Maximum Performance Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Performance Red. Tolerance Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Time Window Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Counter Wraparound Time
       ResourceTemplate(){Register(FFixedHW, 0x40, 0, 1, 0x4)},// Reference Performance Counter Register
       ResourceTemplate(){Register(FFixedHW, 0x40, 0, 0, 0x4)},// Delivered Performance Counter Register
       ResourceTemplate(){Register(SystemMemory, 0x20, 0, 0xA0051004, 0x3)}, // Performance Ltd Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // CPPC Enable Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Autonomous Selection Enable
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Autonomous Activity Window Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Energy Performance Preference Register
       25,     // Reference Performance
       120,    // Lowest Frequency
       800,    // Nominal Frequency
    })
  } // CPU1

  Device(CPU2) {
    Name(_HID, "ACPI0007")
    Name(_UID, 2)
    Name(_CPC, Package()
    {
       23,     // NumEntries
       3,      // Revision
       100,    // Highest Performance
       75,     // Nominal Performance
       25,     // Lowest Nonlinear Performance
       12,     // Lowest Performance
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Guaranteed Performance Register
       ResourceTemplate(){Register(SystemMemory, 0x20, 0, 0xA0051010, 0x3)}, // Desired Perf. Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Minimum Performance Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Maximum Performance Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Performance Red. Tolerance Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Time Window Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Counter Wraparound Time
       ResourceTemplate(){Register(FFixedHW, 0x40, 0, 1, 0x4)},// Reference Performance Counter Register
       ResourceTemplate(){Register(FFixedHW, 0x40, 0, 0, 0x4)},// Delivered Performance Counter Register
       ResourceTemplate(){Register(SystemMemory, 0x20, 0, 0xA0051014, 0x3)}, // Performance Ltd Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // CPPC Enable Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Autonomous Selection Enable
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Autonomous Activity Window Register
       ResourceTemplate(){Register(SystemMemory, 0, 0, 0, 0)}, // Energy Performance Preference Register
       50,     // Reference Performance
```

```
        240,    // Lowest Frequency
        1500,   // Nominal Frequency
     })
   } //CPU2
  } //SCOPE
} //DefinitionBlock
```

Appendix C FFH Operation Regions

C.1 Example of FFH Operation Regions which trigger SMC or HVC calls

The following ASL code shows the use of FFH Operation Region to trigger SMC32 calls using 5 registers.

```
// FFH operation region for using SMC32 calls using 5 registers (FID + 4 args)
OperationRegion(AFFH, FFixedHW, 0, 20)

Field (AFFH, BufferAcc, NoLock, Preserve) { SMCC, 160 }

// Create the command buffer (20 bytes)
Name(BUFF, Buffer(20){}) // Create SMCCC command buffer. Must match the field length.
CreateDWordField(BUFF, 0x00, BFW0) // W0 (FID)
CreateDWordField(BUFF, 0x04, BFW1) // W1
CreateDWordField(BUFF, 0x8, BFW2) // W2
CreateDWordField(BUFF, 0xC, BFW3) // W3
CreateDWordField(BUFF, 0x10, BFW4) // W4

BFW0 = 0x8200FFFF // Set FID. The OS validates that this field is within ranges.
BFW1 = 0xdeadbeef // Some param
...
BUFF = (SMCC = BUFF) // Invoke SMC32 and get return values.
If (BFW0 == 0x0) {
// Success
}
```

The following ASL code shows the use of FFH Operation Region to trigger SMC64 calls using 18 registers.

```
// FFH operation region for using SMC64 calls using 18 registers (FID + 17 args)
OperationRegion(AFFH, FFixedHW, 1, 144)

Field (AFFH, BufferAcc, NoLock, Preserve) { SMCC, 1152 }

// Create the command buffer (144 bytes)
Name(BUFF, Buffer(140){}) // Create SMCCC command buffer. Must match the field length.
CreateQWordField(BUFF, 0x00, BFX0) // X0 (FID)
CreateQWordField(BUFF, 0x08, BFX1) // X1
CreateQWordField(BUFF, 0x10, BFX2) // X2
CreateQWordField(BUFF, 0x18, BFX2) // X3
…
CreateQWordField(BUFF, 0x88, BFX17) // X17

BFX0 = 0xC200FFFF // Set FID. The OS validates that this field is within ranges.
BFX1 = 0xdeaddeadbeef // Some param
...
BUFF = (SMCC = BUFF) // Invoke SMC64 and get return values.
If (BFX0 == 0x0) {
// Success
}
```